Università degli Studi di Salerno
Corso di laurea in Informatica

INGEGNERIA DEL SOFTWARE

"StudyMe"

OBJECT DESIGN DOCUMENT

Docente:
Andrea De Lucia

Studenti:

Nome	***Matricola***
Buono Claudia	0512105296
Esposito Mariarosaria	0512105692
Tripodi Maria Rachele	0512105356

Anno accademico 2019/20

Sommario

1. Introduzione

Dopo la realizzazione dei documenti RAD e SDD abbiamo descritto in linea di massima quello che sarà il nostro sistema e gli obbiettivi da seguire, tralasciando gli aspetti implementativi. Il seguente documento ha lo scopo di definire le interfacce delle classi, le operazioni, i tipi, gli argomenti e le signature dei sottosistemi definiti nel System Design. Inoltre, sono specificati i trade-off e le linee guida.

1.1 Object design trade-offs

- **Prestazioni vs Costi**
  Essendo un progetto universitario e non avendo finanziamenti esterni, si utilizzeranno delle tecnologie open-source in grado di gestire il sistema in maniera gratuita. Per questo motivo i costi non sono un aspetto da tenere in considerazione, mentre per quanto riguarda le prestazioni le tali tecnologie forniscono comunque dei buoni risultati sotto diversi aspetti e per questo rappresentano una buona soluzione per la realizzazione del sistema.
- **Interfaccia vs Usabilità**
  L'interfaccia verrà gestita in modo tale da poter essere il più semplice ed intuitiva possibile, attraverso l'uso di form e bottoni di facile comprensione per l'utente finale.
- **Sicurezza vs Efficienza**
  Il sistema si baserà prevalentemente sulla gestione della sicurezza per evitare accessi non autorizzati cosi da proteggere informazioni personali.

1.2 Interface documentation guidlines

Durante la fase implementativa del progetto gli sviluppatori dovranno seguire le seguenti linee guida:

1.2.1 Naming Convention

I nomi utilizzati per la rappresentazione dei concetti principali, delle funzionalità e delle componenti generiche del sistema devono rispettare le seguenti condizioni:

1. I nomi devono essere:
   a. Di lunghezza medio-breve
   b. Caratterizzati da caratteri compresi in [0-9,a-z,A-Z].
2. Le variabili devono:
   a. Rispettare la Camel Notation;
   b. Iniziare con le lettere minuscole;
   c. Essere composti da caratteri compresi in [0-9,a-z,A-Z].
   Esempio corretto:
   private int nomeVariabile;
3. Le costanti devono:
   a. Utilizzare solamente caratteri maiuscoli;
   b. Separare i vari nomi che le compongono da ("-");
   c. Evitare di iniziare con ("_");
   d. Contenere solamente caratteri[0-9,a-z,A-Z];
   e. Essere di lunghezza medio-breve.
   Esempio corretto:
   private finali nt NOME_VARIABILE;
4. Le classi devono:
   a. Iniziare con la lettera maiuscola;
   b. Rispettare la Camel Notation;
   c. Ogni parola che segue la prima deve iniziare con la maiuscola;
   d. Il nome deve essere un sostantivo;
   e. Evitare di utilizzare acronimi.
   Esempio corretto:
   public class Pacchetto{….}
5. I Package devono:
   a. Contenere solamente caratteri minuscoli;
   b. Contenere solamente caratteri [a-z];

c. Di semantica affine con gli elementi da cui è composto;

6. I metodi devono:
    a. Iniziare con le lettere minuscole;
    b. Rispettare la Camel Notation;
    c. Evitare di iniziare con GET o SET se non si trattano getting o setting della classe corrispondente;
    d. Contenere solamente caratteri [a-z,A-Z].
7. Le pagine JSP:
    a. Contengono solamente caratteri minuscoli.
    b. Sono di lunghezza medio-breve.

1.2.2 Gestione formattazione del codice

Per rispettare i criteri di comprensibilità definiti nella sottosezione precedente, bisogna approcciarsi con cura sull'indentazione del codice e su come alcuni elementi del codice devono essere formattati. Di seguito verranno descritti alcuni esempi per dare una visione generale al programmatore su come si vuole indentare il codice.

1.2.2.1 Gestione codice HTML e XML

```
<html>
    <head>

    </head>

        <body>

            <div>
                Contenuto DIV
            </div>

        </body>
</html>
```

Il codice HTML deve essere intentato in maniera tale da poter mantenere sulla stessa colonna TAG di apertura e di chiusura. Inoltre, contenuto di una distanza pari ad un TAB. i TAG che non hanno una clausola di chiusura seguiranno solamente la seconda condizione.

1.2.2.2 Gestione codice classi Java e Servlet

All'interno di questa sezione ci soffermeremo sulla corretta formattazione della classi Java, delle Servlet e dell'uso di codice Javadoc corrispondente.
Le classi Java devono seguire le seguenti condizioni:

1. Il codice Javadoc deve essere utilizzato per la descrizione di:
    a. Classi.
    b. Metodi.
2. I metodi, le classi interne e le variabili devono essere intentati in colonne successive a quella della classe o del metodo in cui sono contenute.

Esempio:

```java
package esempio;
/**
 *
 * @author Mario Rossi
 * @version 1.1
 * @since   03/12/2019
 */

public class esempio {

    private int x;
    /**
     *  Costruttore della classe
     *  @param x
     *
     */
    public esempio(int x) {
        this.x= x;
    }
    /**
     * Effettua la somma tra le variabili di istanza e il parametro passato.
     *
     * @param int
     * @return int
     */
    public int sum(int y) {
        return this.x=this.x+y;
    }
```

Il codice Javadoc di ogni classe deve contenere la clausola @author,@versione e @since. Mentre ogni metodo deve contenere obbligatoriamente una descrizione delle operazioni o della funzionalità che esegue e può richiamare clausole come @param e @return.

Il formato delle classi Servlet, invece deve:

a. Contenere un costruttore, anche se vuoto;
b. Contenere i metodi doGet() e doPost();
c. Contenere il codice Javadoc per la classe per i metodi doGet() e doPost() in modo da definire lo scope della Servlet e le operazioni dei due metodi.

Esempio:

```java
import java.io.IOException;

/**
 *
 * Descrizione di cosa fa la Servlet
 *
 */
@WebServlet ("/ExampleServlet")
public class EsempioServlet extends HttpServlet {
    private static final long serialVersionUID= 1L;

    public EsempioServlet() {

    }


    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException{
        /*Content of GET*/
    }


    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException{
        /*Content of POST*/
    }

}
```

È possibile aggiungere qualsiasi clausola si voglia all'interno del codice Javadoc della classe e dei metodi. Inoltre, è possibile anche implementare metodi o funzioni proprie della classe in modo da rendere le sue operazioni più modulari.

1.3 Definitions, acronyms and abbreviations

Acronimi:

- **SDD**: System Design Document.
- **ODD**: Object Design Document.
- **RAD**: Requirements Analysis Document.

Abbreviazioni:

- **DB**: Database.
- **DBMS**: Database Management System.

Definizioni:

- **Servlet:** Classi ed oggetti Java per la gestione di operazioni su un Web Server.

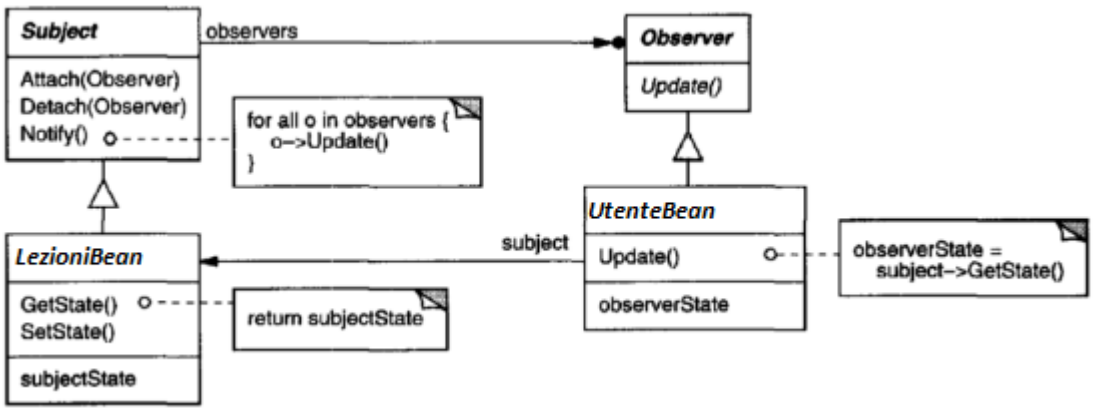
1.4 References

Il contesto è ripreso dal **RAD** e dall' **SDD** del progetto StudyMe.

2. Design Pattern

Utilizzeremo tre tipologie di design pattern:

1. **Observer:** Esso è utile quando al cambio di stato di un oggetto (Subject) un altro oggetto incaricato del suo monitoraggio (Observer), deve essere notificato.
   Nel nostro caso il problema era quello di segnalare all'acquirente dell'inserimento di una nuova lezione in uno dei pacchetti acquistati.
   La soluzione è stata:
   - Far implementare alla classe UtenteBean l'interfaccia Observer che offre il metodo "update()", ovvero l'azione da compiere al momento del cambio di stato.
   - Far estendere alla classe LezioniBean la classe Subject la quale offre il metodo "notifyObservers()" per informare agli observer registrati a quel subject del cambiamento di stato.



2. **Singleton**: Esso ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.
   Nel nostro caso il problema stava nella classe EmailSender, la quale ha lo scopo di inviare email di cambio password o ai cambi di stato(quindi utilizzata da Observer) e creare istanze ad ognuna di queste situazioni risultava costoso.

La soluzione è stata:
- Trasformare la classe EmailSender in un Singleton che abbia un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe. La classe fornisce un metodo getter statico(getIstance()) che restituisce l'istanza della classe (sempre la stessa), creandola preventivamente o alla prima chiamata del metodo, e memorizzandone il riferimento in un attributo privato anch'esso statico.

```
public class EmailSender {

    private static EmailSender instance;

    private EmailSender(){
        this(EmailSender.EMAIL, EmailSender.PASSWORD);
    }

    public static EmailSender GetInstance(){
        if(instance == null)
            instance = new EmailSender();

        return instance;
    }
```

3. **Object Pool Pattern:** ha lo scopo di utilizzare un set di oggetti inizializzati tenuti pronti per l'uso - un "pool" - anziché allocarli e distruggerli su richiesta. Un client del pool richiederà un oggetto dal pool ed eseguirà operazioni sull'oggetto restituito. Quando il client ha terminato, restituisce l'oggetto al pool anziché distruggerlo; questo può essere fatto manualmente o automaticamente.

```
public class DriverManagerConnectionPool  {

    private static List<Connection> freeDbConnections;

    static {
        freeDbConnections = new LinkedList<Connection>();
    }

    private static synchronized Connection createDBConnection() throws SQLException {
        Connection newConnection = null;
        String ip = "127.0.0.1";
        String port = "3306";
        String db = "studyme";
        String username = "root";
        String password = "Mariarosaria)(";

        newConnection = DriverManager.getConnection("jdbc:mysql://"+ ip+":"+ port+ "/"+ db + "?useSSL=false", username, password);
        newConnection.setAutoCommit(false);
        return newConnection;
    }
```

```
    public static synchronized Connection getConnection() throws SQLException {
        Connection connection;

        //Se non è vuota prendiamo la connessione al primo posto e la assegnamo come connessione
        if (!freeDbConnections.isEmpty()) {
            connection = (Connection) freeDbConnections.get(0);
            freeDbConnections.remove(0);

            try {
                if (connection.isClosed())
                    connection = getConnection();
            } catch (SQLException e) {
                connection.close();
                connection = getConnection();
            }
        } else {
            connection = createDBConnection();
        }

        return connection;
    }

    //Questo metodo quando non serve più una connessione la rinserisce nella lista
    public static synchronized void releaseConnection(Connection connection) throws SQLException {
        if(connection != null) freeDbConnections.add(connection);
    }
}
```

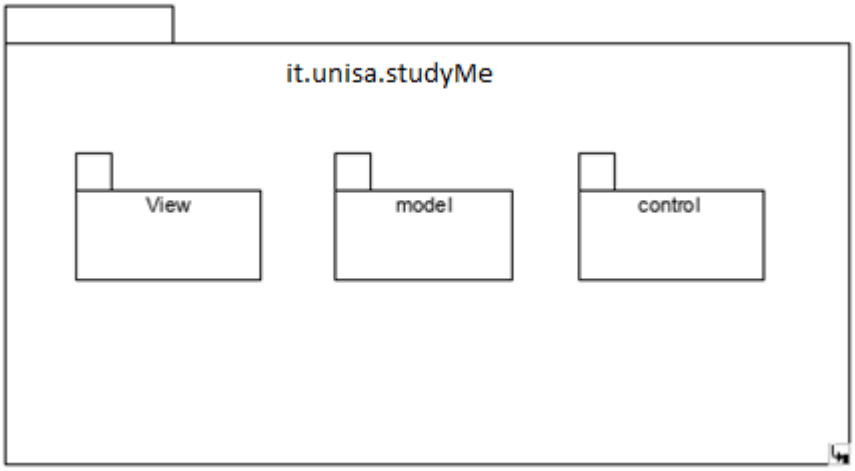
3.Packages

La gestione del sistema è divisa in tre livelli:

View	Si occupa dell'iterazione con l'utente e contiene tutti gli oggetti interagiscono con esso (JSP).
Model	Contiene il dominio applicativo. Si occupa di Gestione Utente, Gestione Corsi, Gestione Lezioni (JavaBean, dao)
Control	Riceve il comando dall'utente (attraverso il View) e li attua modificando lo stato degli altri due componenti (Servlet)

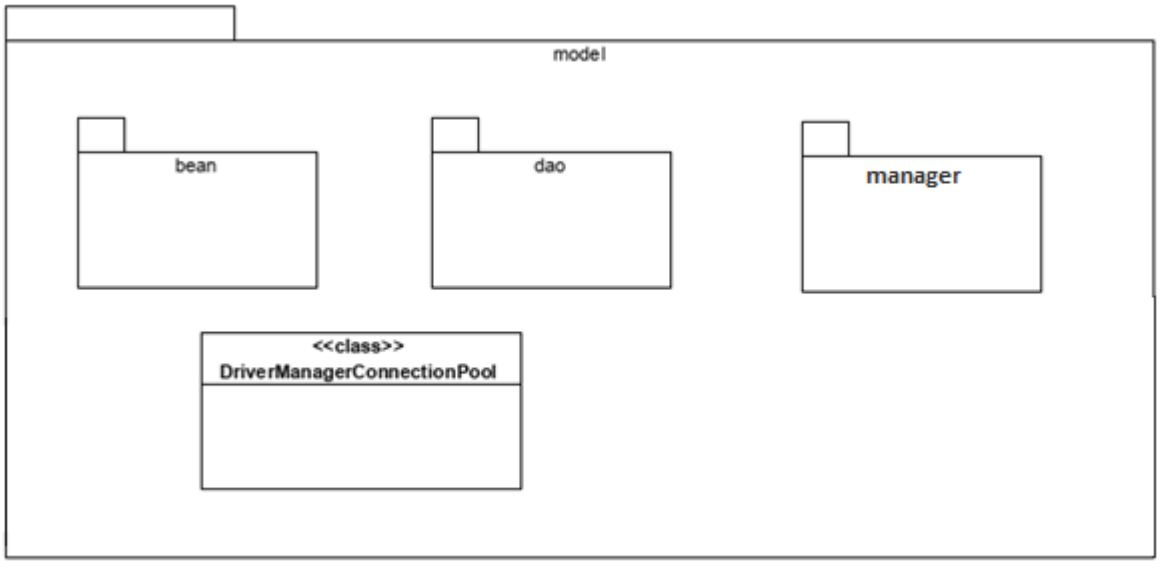
3.1 Package it.unisa.studyMe

Di seguito riporteremo il package core che raggruppa tutti i sottopackage funzionali che verranno utilizzati nella fase di implementazione. Tali sottopackage sono divisi a secondo delle operazioni che eseguiranno all'interno del sistema.



3.1.1 Package Model

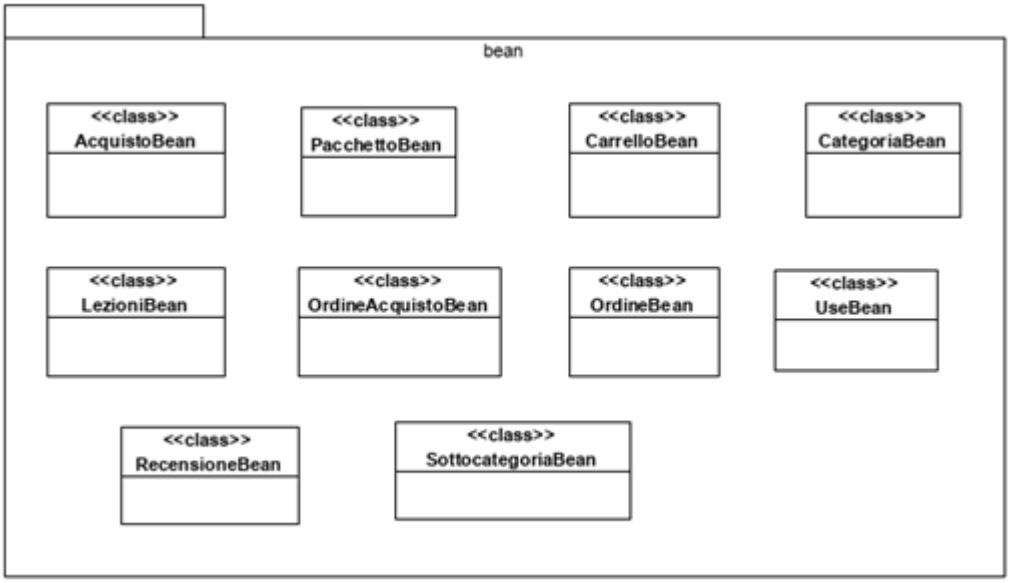
Il Package Model include al suo interno due package (uno per i bean e uno per i dao) e una classe DriverManagerConnectionPool.java



Nome	Descrizione
DriverManagerConnectionPool.java	Permette la connessione al db e il reperimento o la scrittura di dati da/su esso

3.1.2 Package bean

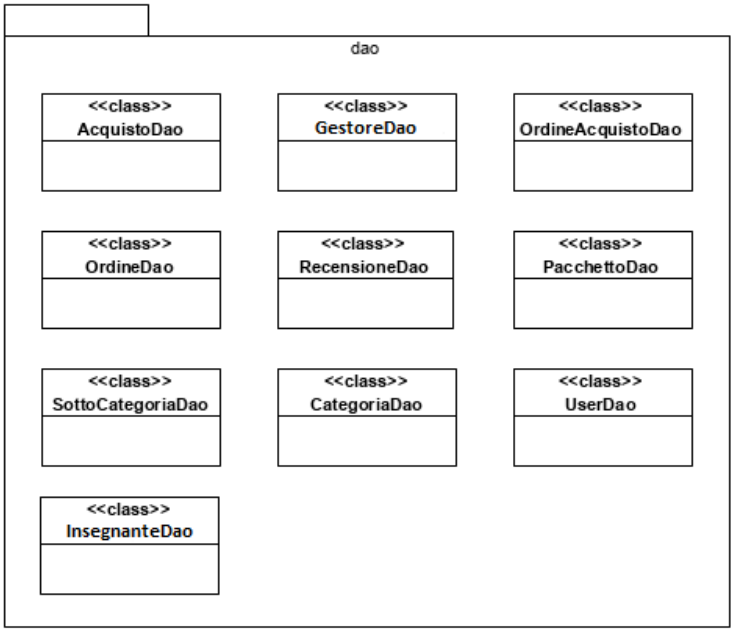
Il package bean include tutte le classi JavaBean. Di seguito, ecco riportato il package che raggruppa tutte le classi del sistema che hanno tale caratteristica:



Nome	Descrizione
AcquistoBean	Descrive un acquisto
CarrelloBean	Descrive un carrello
CategoriaBean	Descrive una categoria
LezioniBean	Descrive una lezione appartenente a un pacchetto
OrdineAcquistoBean	Descrive gli acquisti dei pacchetti di un professore
OrdineBean	Descrive un Ordine nel sistema
PacchettoBean	Descrive un pacchetto nel sistema
RecensioneBean	Descrive una recensione di un utente del sistema associato ad un pacchetto
SottocategoriaBean	Descrive una sottocategoria
UserBean	Descrive un Utente del sistema

3.1.3 Package dao

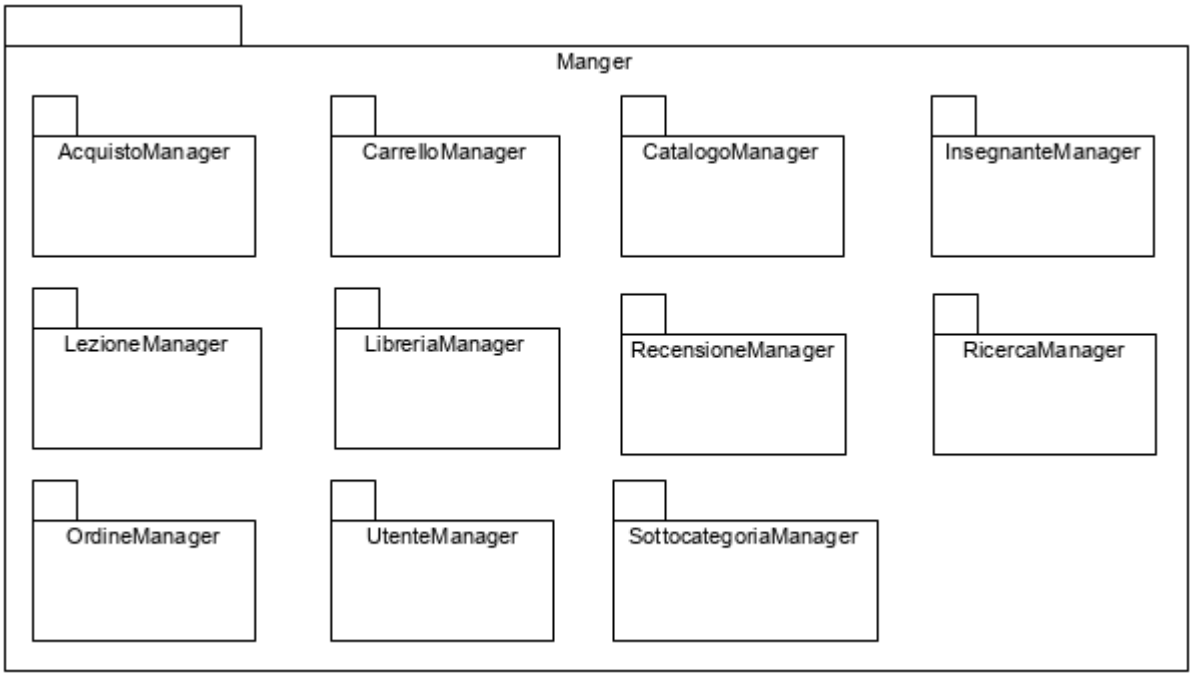
Il package dao include tutte le classi dao. Di seguito, ecco riportato il package che raggruppa tutte le classi del sistema che hanno tale caratteristica:



Nome	Descrizione
AcquistoDao.java	Classe di gestione dati di un acquisto
GestoreDao.java	Classe di gestione dati delle lezioni e dei pacchetti da approvare
OrdineAcquistoDao.java	Classe di gestione dati degli ordini degli acquirenti
OrdineDao.java	Classe di gestione dati degli ordini
RecensioneDao.java	Classe di gestione dati di una recensione
PacchettoDao.java	Classe di gestione dati di un pacchetto
SottoCategoriaDao.java	Classe di gestione dati di una sottocategoria
CategoriaDao.java	Classe di gestione dati di una categoria
UserDao.java	Classe di gestione dati di un utente
InsegnanteDao.java	Classe di gestione dati delle lezioni e dei pacchetti

3.1.4 Package Manager

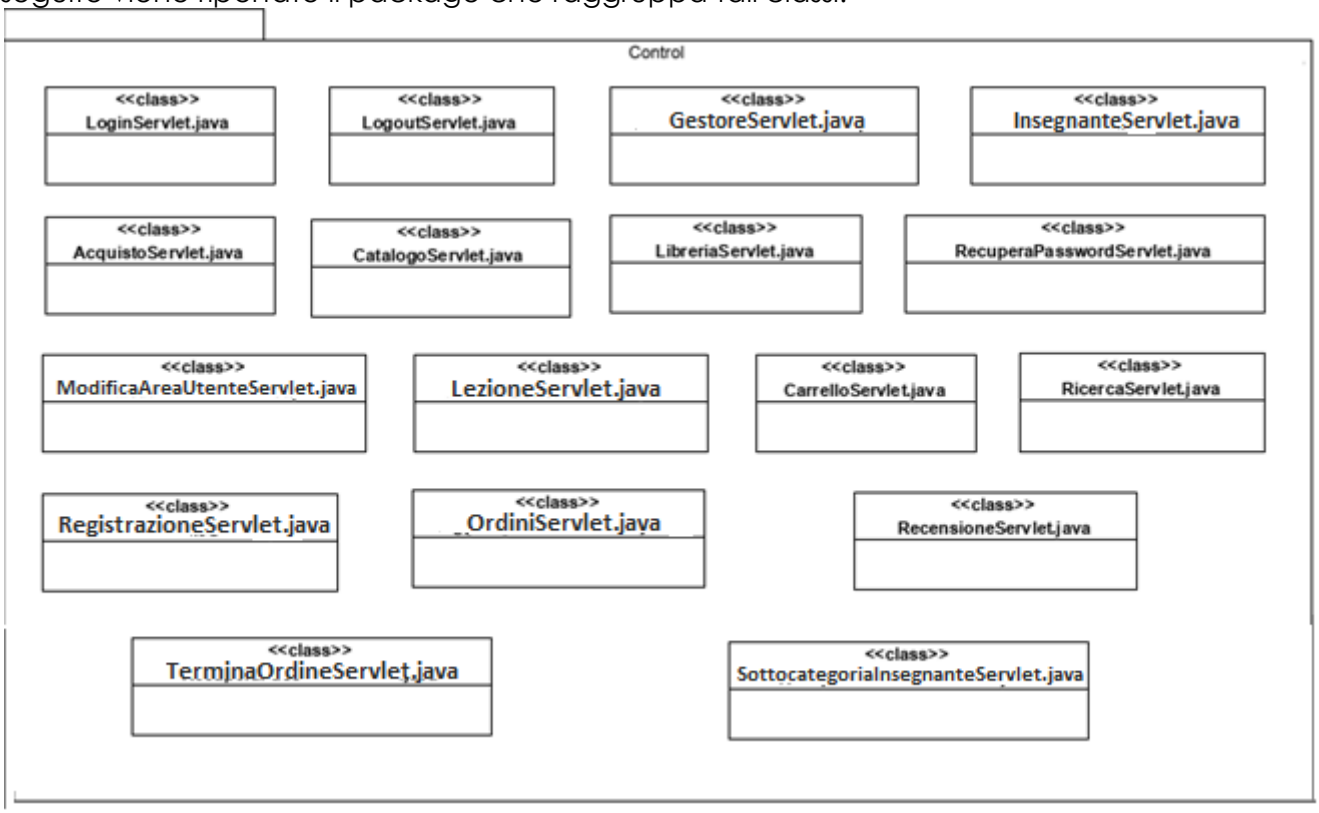
Il package manager include tutte le classi manager si occupa di fornire classi che consentono l'accesso ai dati utili all'applicazione.
Di seguito viene riportato il package che raggruppa tali classi:



Nome	Descrizione
AcquistoManager.java	Classe di gestione di un acquisto.
CarrelloManager.java	Classe di gestione del carrello.
CatalogoManager.java	Classe di gestione del catalogo.
InsegnanteManager.java	Classe di gestione dell'insegnante.
LezioneManager.java	Classe di gestione di una lezione.
LibreriaManager.java	Classe di gestione della libreria.
OrdineManager.java	Classe di gestione degli ordini.
RecensioneManager.java	Classe di gestione della recensione.
RicercaManager.java	Classe di gestione della ricerca.
SottocategoriaManager.java	Classe di gestione della sottocategoria.
UtenteManager.java	Classe di gestione dell'utente.

3.1.5 Package control

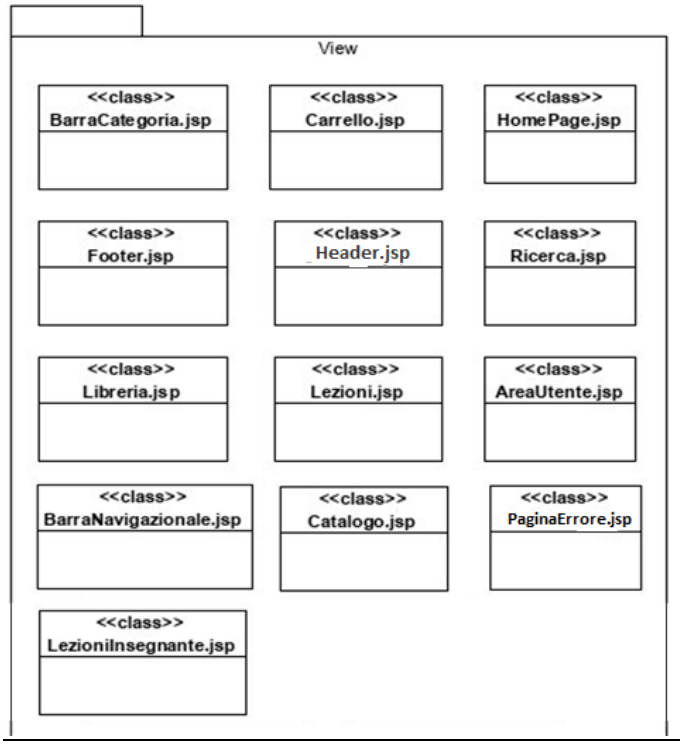
Il package control include tutte le classi Servlet che rappresentano la logica applicativa della nostro piattaforma.
Di seguito viene riportato il package che raggruppa tali classi:



Nome	Descrizione
LoginServlet.java	Controller che permette di completare un'operazione di login.
LogoutServlet.java	Controller che permette di effettuare il logout.
RecuperaPasswordServlet.java	Controller che permette di recuperare la password.
AcquistoServlet.java	Controller che permette di acquistare un pacchetto.
RicercaServlet.java	Controller che permette di ricercare un pacchetto.
GestoreServlet.java	Controller che gestisce l'approvazione e la disapprovazione di un pacchetto.
InsegnanteServlet.java	Controller che gestisce l'inserimento, la modifica e l'eliminazione di un pacchetto e di una lezione.
LezioneServlet.java	Controller che gestisce le lezioni.
LibreriaServlet.java	Controller che gestisce la visualizzazione dei pacchetti acquistati.
ModificaAreaUtenteServlet.java	Controller che gestisce la modifica dei dati personali.
OrdiniServlet.java	Controller che gestisce l'ordine dell'acquirente.
RegistrazioneServlet.java	Controller che gestisce la registrazione di un nuovo utente.
CatalogoServlet.java	Controller che gestisce la visualizzazione dei pacchetti nelle varie categorie.
CarrelloServlet.java	Controller che gestisce l'acquisto di un pacchetto.
RecensioneServlet.java	Controller che permette l'inserimento di una recensione.
TerminaOrdineServlet.java	Controller che gestisce la conclusione di un ordine.
SottocategoriaInsegnanteServlet.java	Controller che gestisce la sottocategoria dell'insegnante.

3.1.6 Package View

Il package view include tutte le pagine JSP, ossia quelle pagine che gestiscono la visualizzazione dei contenuti da parte di un utente del sistema.
Di seguito viene riportato il package che raggruppa tutte le pagine del sistema che hanno tale caratteristica:



Nome	Descrizione
BarraCategoria.jsp	Pagina che include la barra delle categorie principali del sistema.
Carrello.jsp	Pagina che include la lista dei pacchetti da acquistare e la funzionalità per proseguire all'acquisto.
Catalogo.jsp	Pagina principale dei pacchetti, presenta la lista dei pacchetti con le loro descrizioni.
Footer.jsp	Pagina che include il footer delle pagine del sistema.
Header.jsp	Pagina che include l'header delle pagine del sistema.
HomePage.jsp	Pagina principale della piattaforma, presenta le informazioni su quest'ultima.
Lezioni.jsp	Pagina principale del singolo pacchetto che include la prima lezione gratuita e la lista delle recensioni affiliata.
Libreria.jsp	Pagina che include la lista di tutti i pacchetti acquistati dall'utente con le relative lezioni.
BarraNavigazione.jsp	Pagina che include la barra di navigazione principale del sistema.
Ricerca.jsp	Pagina che include la lista dei pacchetti fornita come risultato.
AreaUtente.jsp	Pagina che include una form per la modifica dei dati personali, la funzionalità di logout e per l'acquirente, la lista degli ordini, per l'insegnante la form che permette l'inserimento di un nuovo pacchetto e infine per il gestore del catalogo la lista dei pacchetti da approvare.
LezioniInsegnante.jsp	Pagina che mostra la lista delle lezioni appartenenti ad un singolo pacchetto di un insegnante.
PaginaErrore.jsp	Pagina che contiene i messaggi di errore.